ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

ACELERACIÓN EN GPU DEL PROCESAMIENTO DE IMÁGENES CEREBRALES DE ALTA RESOLUCIÓN

GPU ACCELERATION OF BRAIN IMAGE PROCCESSING

Realizado por
Pablo Sánchez Rodríguez
Tutorizado por
Manuel Ujaldón Martínez
Departamento
ARQUITECTURA DE COMPUTADORES

UNIVERSIDAD DE MÁLAGA
MÁLAGA, OCTUBRE 2015

Fecha defensa:
El Secretario del Tribunal

Resumen:

Durante los últimos años se ha venido demostrando el alto poder computacional que ofrecen las GPUs a la hora de resolver determinados problemas.

Al mismo tiempo, existen campos en los que no es posible beneficiarse completamente de las mejoras conseguidas por los investigadores, debido principalmente a que los tiempos de ejecución de las aplicaciones llegan a ser extremadamente largos. Este es por ejemplo el caso del registro de imágenes en medicina.

A pesar de que se han conseguido aceleraciones sobre el registro de imágenes, su uso en la práctica clínica es aún limitado. Entre otras cosas, esto se debe al rendimiento conseguido.

Por lo tanto se plantea como objetivo de este proyecto, conseguir mejorar los tiempos de ejecución de una aplicación dedicada al resgitro de imágenes en medicina, con el fin de ayudar a aliviar este problema.

Palabras claves:

CUDA, GPU, IMAGEN, REGISTRO, RENDIMIENTO

Abstract:

During last few years it has become clear the high computational power offered by GPUs when certain problems are presented.

At the same time, there are fields in which it is not possible to completely take advantage of improvements achieved by reserachers, mainly because execution times become extremely long. This is for example the case of medical image registration.

Although speedups have been achieved on image registration, their use in clinical practice is still limited. Among other things, this is due to the performance achieved.

Thus, it is suggested as the aim of this project to accomplish execution time enhancements for an image registration application, helping to relieve this problem.

Keywords:

CUDA, GPU, IMAGE, PERFORMANCE, REGISTRATION

Contents

1 The GPGPU movement **1**

1.1 The GPU Streaming Processor 1

1.1.1 Advantages and drawbacks 1

1.2 Evolution to a general purpose architecture 2

1.2.1 Starting point 2

1.2.2 GPGPU first steps 3

1.2.3 The arrival of CUDA 4

1.2.4 OpenCL 5

1.2.5 Last years and the future of GPGPU 6

2 Programming with CUDA **9**

2.1 CUDA (Compute Unified Device Architecture) 9

2.2 Programming model 10

2.2.1 Processing levels 10

2.2.2 Streams 11

2.2.3 Processing flow 11

2.3 Hardware model 12

2.4 Evolution of the architecture by generations 13

2.4.1 The first generation: Tesla (G80 and GT200) 13

2.4.2 The second generation: Fermi (GF100) 15

2.4.3 The third generation: Kepler (GK110 y GK210) 16

2.4.3.1 Dynamic Parallelism 18

2.4.3.2 Hyper-Q 19
2.4.4 The fourth generation: Maxwell (GM204) 19
2.4.4.1 Memory improvement 20
2.4.4.2 Shared memory atomics 20

3 Speeding up the code **23**
3.1 Introduction 23
3.1.1 Image Registration 23
3.1.1.1 Operating mode 23
3.2 Starting point for optimizations 24
3.3 Adding GPU support to RNiftyReg 25
3.4 Optimizing NiftyReg 25
3.4.1 First iteration: Identifying the problem 25
3.4.2 Second iteration: reg_getNMIValue() optimizations 27
3.4.2.1 Histogram smoothing 27
3.4.2.2 Histogram smoothing (a failed alternative) 29
3.4.2.3 Histogram normalization 30
3.5 About the achieved results 30

4 Conclusions **35**
4.1 Conclusiones 36

A Appendix **37**

The GPGPU movement

1

1.1 The GPU Streaming Processor

Graphics Processing Units (GPU) were conceived as a processor dedicated to graphics. That is a piece of hardware which frees the CPU from tasks related to graphic processing. One of the reasons for the existence of this dedicated graphics processor is the high computational cost of these tasks, due to the large amount of data to be processed in short time intervals.

Since its inception, the CPU, based on the Von Neumann architecture, has given more importance to the instructions that manipulate data than to the data itself. Because of that, processors are not efficient when accessing to multiple data simultaneously.

The high performance offered by the GPU versus CPU is due to a large change in the way information was handled historically, from a sequential pattern, to a new data-centric model. In this new model, data were grouped into streams, and it was possible to perform calculations on each of their elements at the same time.

The model came as a programming paradigm and resulted in the development of a processor specialized in streams, which was referred to as a Streaming Processor.

1.1.1 Advantages and drawbacks

The operation of the GPU processor-based streaming is what has mainly defined its advantages and drawbacks.

Its main advantage is scalability, that is, the ability to handle a growing amount of work in a capable manner. Since this benefit is based on its architecture, the expectations for the future are very high. For this, the GPU performance doubles every six months, much faster than the CPU.

However, we have to point out that not all applications benefit from its archi-

tecture. On one hand, we have applications which are hard to parallelize; and on the other hand, some of them make heavy use of selection structures in their algorithms.

1.2 Evolution to a general purpose architecture

Over the past few years, it has been increased the use of GPUs to speed up codes that originally ran on a CPU. This change was mainly due to the evolution of GPUs from its original approach (rendering graphics) to a flexible and programmable computer (General Purpose GPU or GPGPU).

Despite being a relatively recent technology, it is having a great acceptation, firstly due to the continuous evolution of GPUs to the GPGPU, and secondly by the results obtained against the CPU and its future expectations.

Since the arrival of the first graphic platforms, a number of improvements have followed to build more efficient devices. So in the following sections we are going to examine in a deeper way the most important stages of this evolution.

1.2.1 Starting point

Since its inception, the GPU has executed parallel algorithms. However, these were responsible for the different stages of the rendering process (graphics pipeline), so these were fixed.

During the 90s, it began to normalize programming GPU since the boom of graphical programming gave birth programming interfaces (including OpenGL and later DirectX). They allowed developers to work with the GPU in a more transparent and efficient manner.

While the software was evolving, hardware companies also modified the graphics pipeline introducing two programmable processors called shaders, making more versatile GPUs. However, these processors were programmed in assembler, so it was necessary for popularizing the emergence of new tools to make more simple their programming.

Thus, in 2002 was born HLSL (High-Level Shading Language) as an initiative of Microsoft. It was a language of a higher level of abstraction than the assembler, but it required the programmer to know the GPU architecture.

Thereafter, in late 2002, appeared Cg (C for graphics). It was developed by Nvidia in collaboration with Microsoft and was very similar to HLSL. The language was

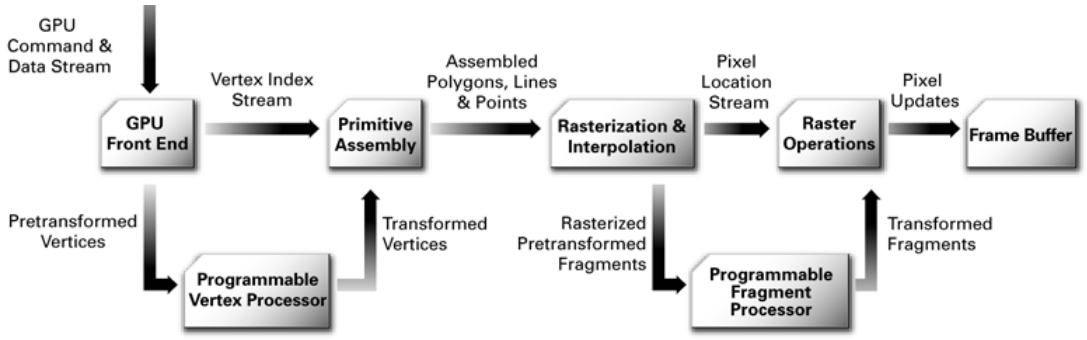

Figure 1.1: *Graphics pipeline after shaders inclusion.*

based on C programming language with elements adapted to GPUs. Faced with HLSL, Cg had all the features of a high level language, more functions to the programmer, and also made the code less dependent on hardware.

Finally, GLSL (OpenGL Shading Language) appeared as an alternative of the OpenGL Architecture Review Board. Also based on C, allowed developers to make cross-platform applications that took advantage of most of the new features of GPUs. It was initially introduced as an extension to OpenGL 1.4, and officially included in version 2.0 of OpenGL in 2004.

1.2.2 GPGPU first steps

At the beginning of this century, GPUs were incredibly programmable. However, until then they had only been used for programming graphics applications.

It was in the scientific sphere when seeing the power of GPUs, tried to compute more general-purpose applications. From the conventional implementation of an algorithm for CPU, a GPU algorithm needed a rewriting to structure input data, instructions and operators to the geometry of a spatial problem. That way the problem was able to be computed by the programmable graphics processors.

Unfortunately, developers must check that no side effects or changes occurs within the graphics pipeline, as it was not designed for this purpose. These tasks required knowledge of the internal architecture, with sufficient skills and previous experience.

Since 2003, we started to see codes taking advantage of GPUs performance. These programs made clear the difference between the CPU and the GPU, which would increase in coming years as developers gained experience and improved their

Algorithms	Improvements
Particle systems Physic simulations Molecular dynamics	2-3
Database queries Data mining Reduction operations	5-10
Signal processing Volume rendering Image processing Biocomputing	10-20
Raytracing 3D visualization	+20

Tabla 1.1: *Improvement when executing different kinds of parallel algorithms.*

algorithms. Table 1.1 shows the differences that were observed.

1.2.3 The arrival of CUDA

In 2003, a team of researchers from outside NVIDIA and led by Ian Buck announced the first programming model that allowed to develop on a GPU using a high level language as if it were a general purpose processor. This not only meant facilities when developing code, but also improved performance.

NVIDIA knew his incredibly fast hardware should be accompanied by a software that were at the cutting edge, so they invited the team to join the company to start developing the next big step for the company. As a union of hardware and software, NVIDIA released CUDA in 2006 as the first global solution for general purpose computing on GPUs. Some of the improvements were:

- Code readability.
- Easy to program and shorter development time.
- Easy to debug and optimize code.
- Independent code of the GPU.
- Complex mathematical operations and accurate results.

CUDA computing platform provided developers with a system based on C/C++ along with several extensions that allowed programmers to implement parallel ap-

	2008	2015
CUDA GPUs	100.000.000	600.000.000
Supercomputers in top500.org	1	75
University courses	60	840
Scientific articles	4.000	60.000

Tabla 1.2: *Evolution of CUDA.*

plications. It also offered alternatives that gave programmers the ability to express parallelism using other high level languages (Fortran, Python ...) and open standards (such as OpenACC directives).

The release of CUDA was widely accepted by scientific, academic and developer communities in general. And the NVIDIA paradigm brought a number of improvements that eliminated all the difficulties encountered. In fact, since its arrival day to today, the CUDA platform has been used in more than 600.000.000 GPUs and 60.000 research applications (see 1.2).

1.2.4 OpenCL

At the end of 2008, OpenCL was released as an open alternative to proprietary solutions for GPGPU. OpenCL was the product of many years of development by an open software consortium. It was originally conceived by Apple and developed in conjunction with AMD, IBM, Intel and NVIDIA; took the realy the Khronos Group and converted into an open, royalty-free standard.

Unlike CUDA, OpenCL is defined as a general purpose programming standard in heterogeneous systems that can run on different architectures, such as CPUs, GPUs and FPGAs. OpenCL provides an API for parallel computing and a programming language based on ISO C99 with extensions for data parallelism.

The operation of OpenCL is based on a host machine that distributes the workload between all devices in the system, which can be one or more computational units. The latter is divided into multiple processing elements.

Although OpenCL is a valid alternative to CUDA, the distance between both is sometimes tremendous. If the implementation and distribution of work is perfectly adjusted to the target architecture, OpenCL performance should not be much less than CUDA. However, here is the key, since the main feature of OpenCL is portability.

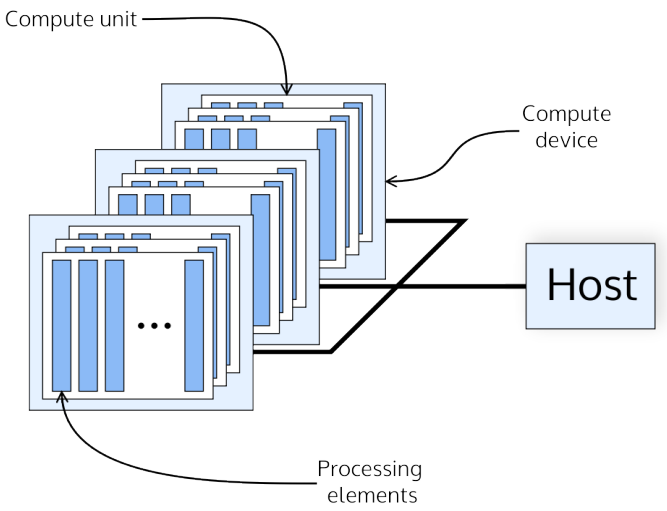

Figure 1.2: *The OpenCL model.*

	June 2011	June 2012	June 2013	June 2014
NVIDIA Fermi	12	53	31	18
NVIDIA Kepler	0	0	8	28
Intel Xeon Phi	0	1	11	21
ATI Radeon	2	2	3	3
IBM Cell	5	2	0	0
Hybrid	0	0	1	4
Total	19	58	54	74

Tabla 1.3: *Evolution of GPUs in TOP500.*

1.2.5 Last years and the future of GPGPU

The programming of GPUs has evolved a lot in recent years. However, knowing its evolution, the following step was obvious, to increase scalability out of the GPU itself.

To do that, clusters of computers arise and more devices interconnect, which operate in groups acting as one graphics device, and this led to emergence the GPGPU movement to gain momentum in the field of high performance computing.

The enhancement was not limited only to the appearance of servers and workstations, but at the same time allow to raise the number of heterogeneous supercomputers that incorporated latest generation GPUs as coprocessors to carry out part of the work. The table 1.3 shows the evolution of graphics coprocessors in the TOP500 supercomputers list in the last four years.

The change to the GPGPU model is relatively recent, so it still has a long way to go. GPUs offer performance several orders of magnitude greater than the CPU so they are positioned as an alternative to traditional processors and could be considered as the computing engine for the future.

Programming with CUDA

2

2.1 CUDA (Compute Unified Device Architecture)

CUDA[20] is a parallel computing platform and programming model designed by NVIDIA that allows developers to access the computing power of GPUs to solve data-parallel problems. The model is composed of three different levels:

- **Software:** On the software level, the CUDA model offers a set of different ways to develop applications and write code to be run on GPUs. Among them we can find:
  - **Programming APIs:** Allow developers to implement GPU code in their programming language of choice. Although C/C++ are the most common high-level languages to develop CUDA applications, there exist APIs for other languages such as Fortran, Java or Python.
  - **Optimized libraries:** There are several libraries prepared so that developers can make full use of them with just a few lines of code, allowing them to make use of GPU-acceleration. (cuBLAS, cuFFT, Thrust, etc.)
  - **Compiler directives:** Standard compiler directives, like those from OpenACC, simplify code acceleration by only requiring prorgrammers to identify code sections that can exploit data parallelism.
- **Firmware:** NVIDIA offers a computing driver that is compatible with the one responsible for rendering. This driver can be controlled through simple APIs to manage CUDA devices, video memory and other components of the architecture.
- **Hardware:** CUDA is implemented so that applications can be run on different compatible hardware implementations. This point is explained in detail in section 2.3.

2.2 Programming model

Next, the CUDA programming model is presented in its C/C++ version. It is an extension for the C language that serves as an interface for parallel programming on GPUs.

In this model, the GPU acts like a coprocessor and only executes a fraction of the code, while the rest is handled by the CPU. In order to be able to work in this way, the CUDA compiler (NVCC) has to separate device (GPU) code from host (GPU) code:

1. **Device code** compiles to *PTX*, a low-level instruction set. It is compatible with diferent devices, allowing the developer to avoid the details of the particular hardware implementation.
2. **Host code** is sent to C compiler, just like any other application code, and allows the program to communicate with the GPU drivers.

Finally, the linker produces a CPU-GPU executable. For NVCC to be able to divide the code, it is necessary to introduce new syntax elements that are used by the programmer to define kernels, CUDA-C functions which contains the code to be executed in each GPU thread.

2.2.1 Processing levels

In order to be used from CPU code, kernels have to be declared as `__global__`. To launch a kernel, host code must include a declaration similar to `KernelNameToLaunch«<G, B, m, s»>`, with **G** and **B** being grid and block sizes, **m** being shared memory size and **s** being the stream to be used. We will describe these arguments more thoroughly, starting with G and B. Threads are identified inside kernels as follows:

1. **Threads are organized in blocks**. Each thread has an identifier that is accessible within the kernel through the built-in variable `threadIdx`.
2. **Blocks are grouped into a grid** and, like threads, to each block is also given a unique identifier, `blockIdx`.

Both grid and thread blocks can be 1D, 2D or 3D, and their sizes are set by the programmer under certain constraints. Their dimensions are accessible within the kernel through the variables `blockDim` and `gridDim` respectively. This allows CUDA code to be scalable by being able to run on any compatible hardware without a need to recompile for different target sizes.

In addition to configurable grid and block sizes, threads are also grouped into warps, that up to this day contain 32 threads. They are the **minimal processing unit**, and are executed in random order although they can be synchronized if required.

Warps execute one common instruction at a time for all their threads. Because of this, warp divergence caused by branching is very costly in terms of performance. If such divergence happens, only threads in the same brach are running simultaneously. When all execution paths complete, the threads converge back. This serialization only occurs within a warp, and never happens when two different warps are able to execute distinct paths at the same time.

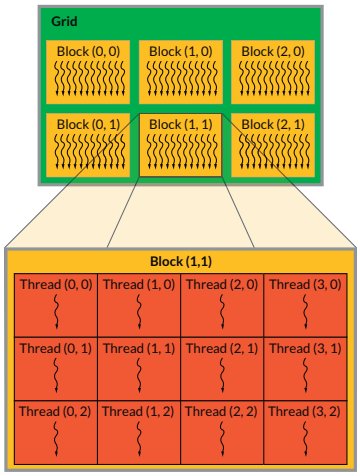

Figure 2.1: *Graphical representation of a grid with six thread blocks each one of 12 threads.*

In the same way as threads, blocks are also executed in random order, but they cannot be synchronized. Threads are able to comunicate only with others threads in the same block using shared memory.

2.2.2 Streams

Since the second generation of CUDA, concurrent execution of different kernels has been made posible using streams. A stream is a sequence of kernels that execute in order.

In spite of kernels in different streams are able to execute independently, by default, all the kernels are executed in the same stream. To allow this behaviour, it is mandatory to assign manually each kernel to a different stream through the fourth parameter of the kernel call interface, **s**, we previously saw.

2.2.3 Processing flow

As already mentioned in section 2.2 with CUDA the GPU (device) acts as a coprocessor for the CPU (host) but with its own memory. Because of this, it is necessary to transfer data from host memory to device memory, perform the computation and bring the databack [5].

Although this schema is still in use, future generations will simplify it by adopting an unified memory architecture for both: host and device.

2.3 Hardware model

The massively parallel threading model is built upon the CUDA hardware model. Each generation, the model is expanded upon while backwards compatibility is maintained.

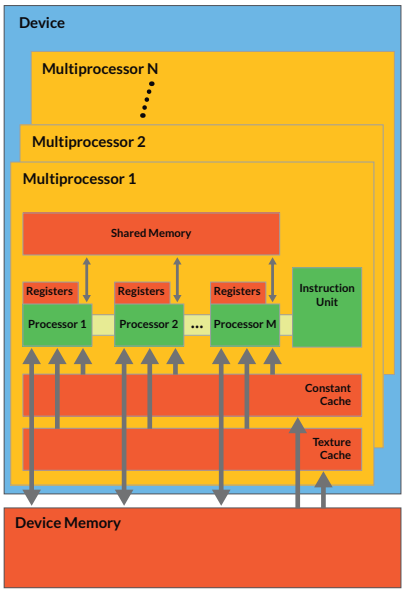

Figure 2.2: *CUDA hardware model.*

The NVIDIA GPU architecture is based on SIMT (Single-Instruction, Multiple- Thread) processing, similar to SIMD (Single Instruction, Multiple Data) but different, because it specifies the execution and branching behaviour of each single thread. This is achieved in hardware by an array of Streaming Multiprocessors (SMs) where each of them contain lots of CUDA cores.

All these cores are conected through a memory hierarchy. In order from faster to slowest are:

- **Registers:** fastest and used by cores to perform most of their computing work.
- **Shared memory:** slightly slower than registers, shared among threads in the same block and used as a cache memory managed by the programmer.
- **Read-only memory:** used for constants and symbols.
- **Global memory:** the slowest, but with the advantage that it is common to all multiprocessors. This memory, of a SGRAM (Synchronous Graphics Random Access Memories) nature, is three times fastest than CPU RAM. However, it is still 500 times slower than shared memory.

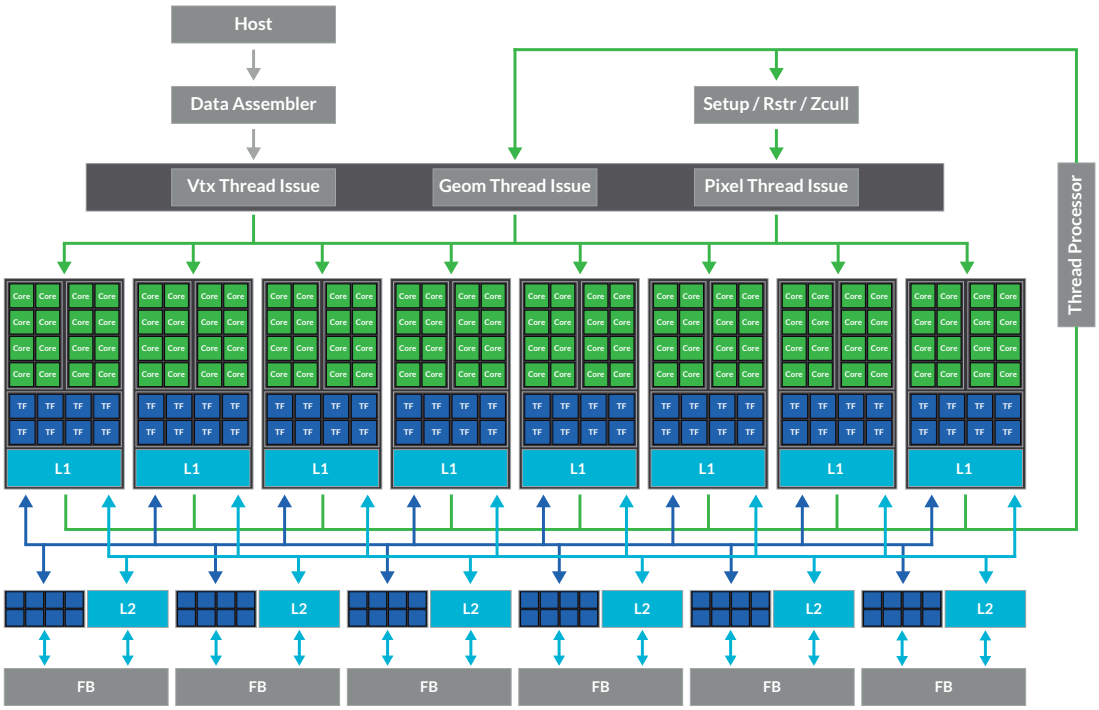

Figure 2.3: *GeForce 8800 GTX (G80) block diagram.*

2.4 Evolution of the architecture by generations

To identify the different architecture models, NVIDIA assigns a version number to each device generation. This number, called CUDA Compute Capability (or C.C.C.), is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

The main features of the different generations are explained below.

2.4.1 The first generation: Tesla (G80 and GT200)

Tesla was the first CUDA capable GPU generation, launched in 2006. It unified the vertex shader with the pixel shader, and allowed them to be used for GPGPU by changing the pipeline going over from a lineal to a loop pipeline.

Each G80 GPU has 8 Thread Processing Clusters (TPC), which in turn have two SMs with 8 cores each. This means that there are 128 scalar processing cores, that

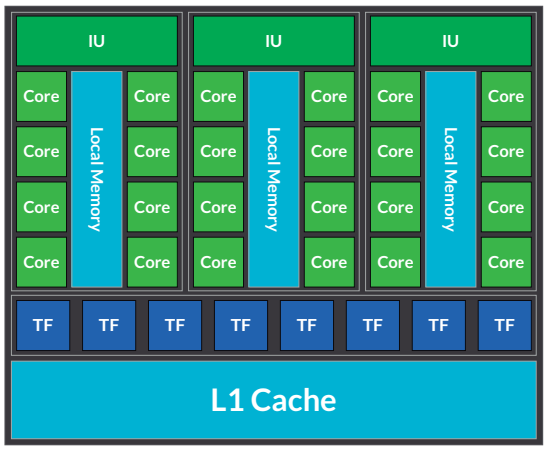

Figure 2.4: *Thread Processing Cluster of GT200*

in addition support dual-issuing MAD and MUL operations. G80 GPUs have 8K 32bit registers and 16Kb of shared memory per SM. Figure 2.3 shows a diagram of the architecture.

Beyond this, NVIDIA improved the Tesla architecture with the GT200 GPU. The main enhancements are listed below:

- **A rise in the amount of cores.** The number of TPC blocks was raised from 8 to 10, with an increase in the amount of SMs per TCP to three. Due to this, GT200 GPUs had 240 cores.
- **More threads per chip.** The software limitation on G80 only allows 768 threads per SM whereas the GT200 accepts until 1024 threads.
- **Doubled register file size.** The register bank is doubled to the previous architecture increasing to 16K registers per SM.
- **Double-precision floating-point support.** One core for fp64 operation is added in each SM.
- **Shared memory improved.** Hardware memory access coalescing was added to improve memory access efficiency.

In the Figure 2.9 is visible the three SMs inside a TCP of a GT200 revealing that in this ocasion the increase of cores is produced by mean of a rise of TCP units instead enlarge the number of cores per SM.

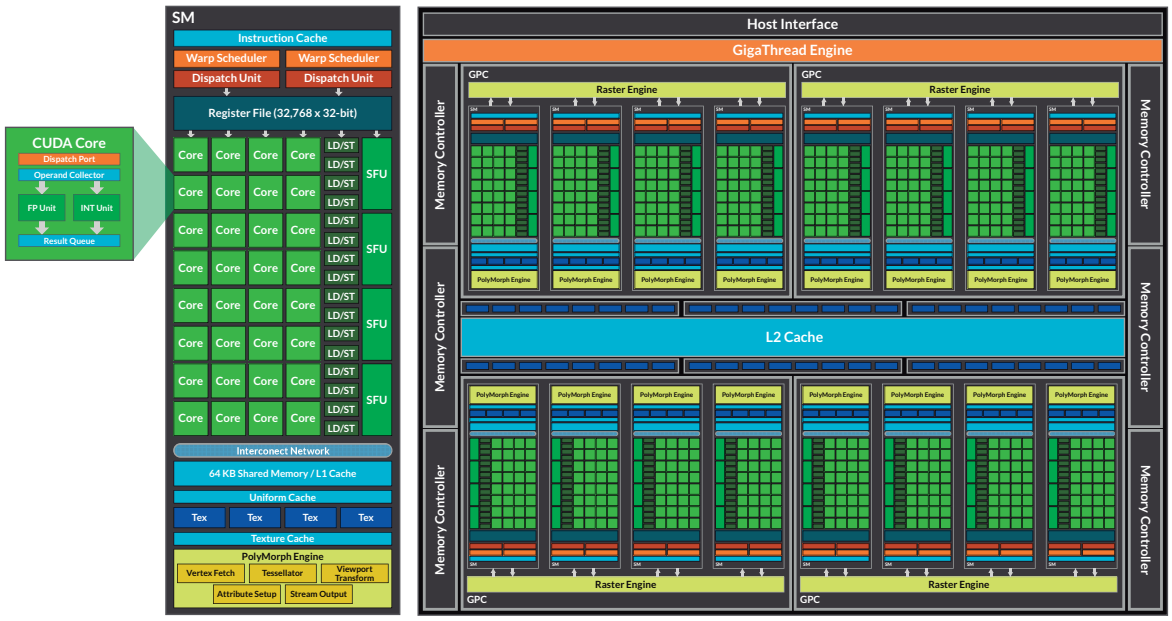

Figure 2.5: *GF100 block diagram and Stream Multiprocessor detail.*

2.4.2 The second generation: Fermi (GF100)

The TCP disappears and Nvidia makes a new hardware block, called Graphics Processing Clusters (GPC), that encapsulates all key graphics processing units. Inside of this hardware block there are four stream multiprocessors.

In this case, NVIDIA decided to reduce the number of SMs and increase the number of cores per multiprocessor. Thus, Fermi has three distinct type of cores:

1. **Int and floating points units**. 32 cores per SM redesigned for optimize 64-bit int operation. These cores are used for both simple and double precision.

2. **Load/Store units**. For Load/Store operations 16 cores are incorporated allowing source and destination addresses to be calculated for sixteen threads per clock.

3. **Special Functions Unit (SFU)**. Four cores are added for quick calculation of complex functions such as sin, cos, reciprocal and root although losing accuracy.

In addition the GF100 has two warp schedulers with an instruction dispatch unit each one. This configuration allows to launch two warps concurrent and inde-

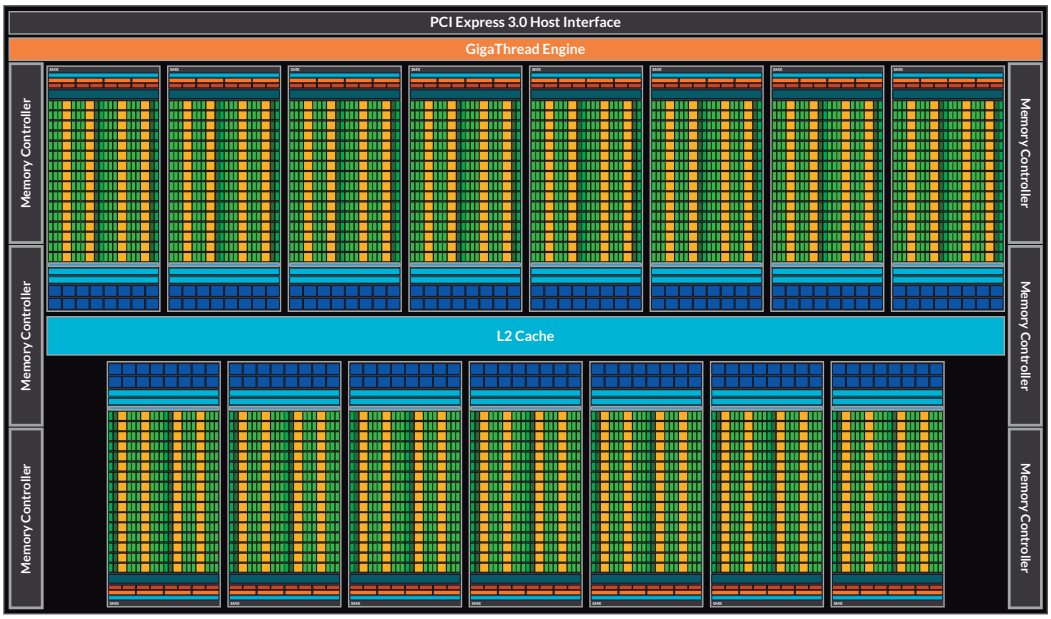

Figure 2.6: *Kepler GK110 full chip block diagram.*

pendently, due to this the schedulers do not need to check for dependencies from within the instruction stream.

One of the main improvements over the previous generation is the memory hiearchy. Each Fermi's SM have 64KB of on-die memory that it is configurable in two mode: 16KB of shared memory and 48KB of L1 cache and vice versa. The first mode optimize the algorithms where data addressees are not known beforehand while the second is the best mode for algorithms with well defined memory access. Moreover this generation incorporates 768KB of L2 cache common to all stream processors.

2.4.3 The third generation: Kepler (GK110 y GK210)

Following the way taken on Fermi, Kepler increases the number of cores per SM and reduced the amount of multiprocessors. Even though the GK110 is not the first chip with Kepler architecture this section is centered in the K110 and higher because they are the most used on servers present.

The quantity of cores per SM is the same in the distinct incremental improvement to the architecture, although the number of stream multiprocessors changes from one to another. Thus, the **??** show the differents versions and its main features.

The Kepler's SMs or SMX have **192 single precision CUDA cores**, and each core has fully pipelined floating-point and integer arithmetic logic units. In addition,

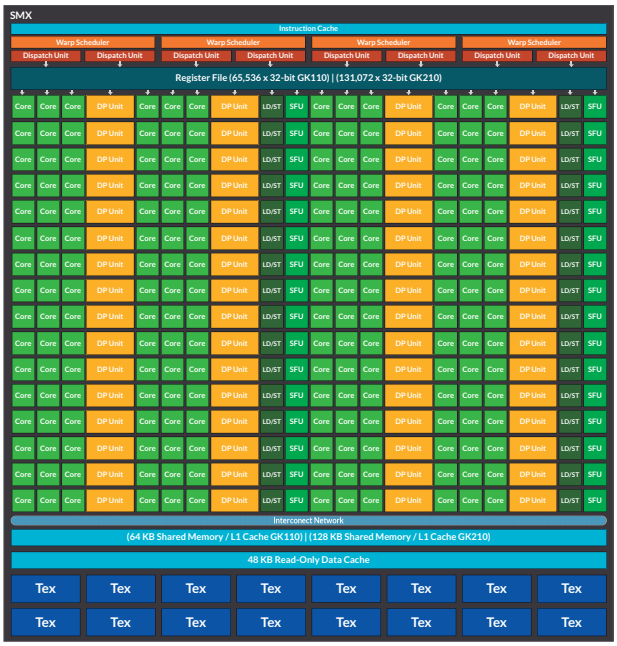

Figure 2.7: *SMX with 192 single-precision CUDA cores, 64 double-precision units, 32 SFU and 32 LD/ST units.*

these SMs increase the **double-precision** computation capacity with **64 dedicated units**. More the GK110 have **32 LD/ST units**,double the amount of load and store units available in the Fermi architecture. Finally, the SMXs have **32 special function units** (SFU).

For load warps each SMX have **four warp schedulers with two dispatch instruction** each one. This allows four warps are issued and executed concurrently even the double precision operations.

In the side of memory, Kepler continues the hierarchy initiated on Fermi although the **texture memory** now is accessible for GPGPU as **only-read memory of 48KB**. In addition this generation improve all blocks of memory:

- **Register Bank**. The amount of 32-bit register per multiprocessor grows until **64K**.
- **Shared Memory and L1 cache**. Besides of the two configuration modes of shared memory were seen in the Section 2.4.2, a **new mode** is added in this generation: **38KB for both**.
- **L2 cache**. The amount of memory in this block is doubled compared with Fermi, reaching **1536KB**. In addition the L2 cache on Kepler offers up to **2x of the**

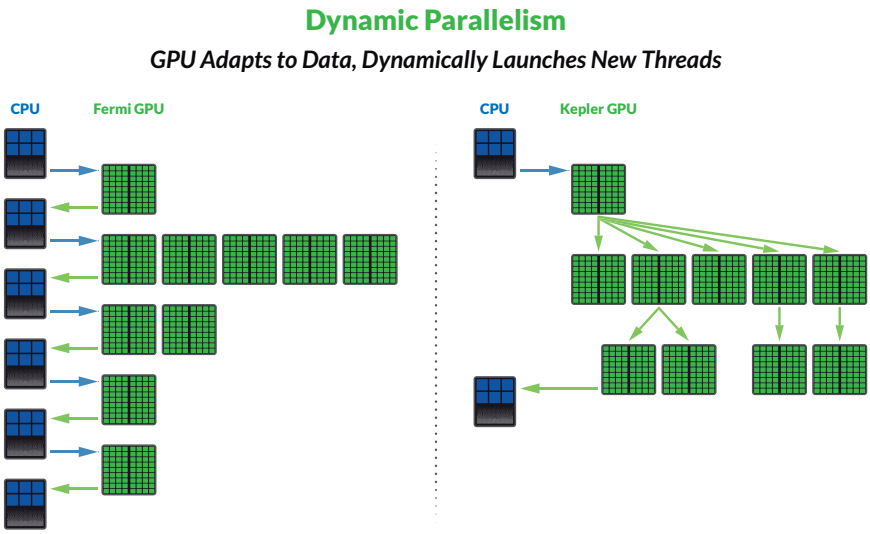

Figure 2.8: *With Dynamic Parallelism the GPU can generate new work for itself.*

bandwidth per clock available on Fermi.

The **GK210** is very similar to GK110 and they both have the features explained in the Section 2.4.3.1 and the Section 2.4.3.2. One as much as the other are kepler architectures but the GK210 has more resource on-chip than its predecessor, the GK110. Thus, both chips share the **same amount of core per SMX** but the GK210 have **128K register of 32-bit per SMX** and **128KB of shared memory/L1 cache** with the configurations below:

- 112KB shared memory + 16KB L1 chache
- 96KB shared memory + 32KB L1 chache
- 48KB shared memory + 80KB L1 chache
- The anterior amounts reversed.

2.4.3.1 Dynamic Parallelism

Until the GK110 was created, only the CPU sends kernels to the GPU. When a kernel needed the result of other kernel to be launched was necessary that the CPU sended the first kernel to GPU, the GPU returned the result to CPU and this last launched the second kernel. Now, with **dynamic parallelism** this process is simplified due to **the GPU can generate new work for itself**, it does not need to interrupt to the

CPU[19]. The Figure 2.8 shows a example about how functions the dynamic parallelism, releasing of work to the CPU.

This new feature allow the programmer to use recursive techniques for its algorithms. Due to this, the developer is able to make **algorithms that were impossible to achieve on FERMI** such as quicksort, nested loops with differing amounts of parallelism or even dynamically setting up a grid for a numerical simulation focusing in the interesting zones without an expensive pre-processing.

On Fermi, the host send a grid to the CUDA Work Distributor (CWD) and this distributes the blocks among the differents SM. On Kepler, is necessary a new unit for **management both the device and host grids**. This component, called **Grid Management Unit (GMU)**, process the grids receives from CPU and GPU and sends to CWD the work. Then the work distributor, that admits until 32 grids, sends the blocks to the SMX. In addition the GMU can pause the dispatch of new grids due to the two-way link.

2.4.3.2 Hyper-Q

On Fermi until 16 streams could be launched concurrently, although they were not executed simultaneously because they were put in a unique queue, only the end of a stream and the start of other were executed at the same time. On Kepler until **32 streams can be really executed concurrently** due to the fact that each stream receives its own work queue. Of this form the programmer makes the most of the GPU computacional capacity.

2.4.4 The fourth generation: Maxwell (GM204)

The new generation of GPUs is focused on maximize the performance per watt consumed. Thus, NVIDIA has reorganized internal components of multiprocessors (SMMs) and now these are splited in four part. Each CUDA cores processing block contains:

1. **32 int and floating points units** (128 per SMM).
2. **1 double precision units** (4 per SMM).
3. **8 Load/Store units** (32 per SMM).
4. **8 Special Functions Unit (SFU)** (32 per SMM).

Figure 2.9: *Maxwell GM204 full chip block diagram.*

In addition to this, each split contains a warp scheduler, which is capable of dispatching two instruction per warp every clock cycle. This configuration aligns with warp size, making it easier to utilize efficiently.

2.4.4.1 Memory improvement

The memory hierarchy is changed too, now the shared memory don't share the block with the L1 cache. The size of shared memory grows to 96KB although it is limited to 48KB per thread block[8]. Finally, the size of L2 cache is 2MB on GM204.

Another improvement implemented on Maxwell is the memory compression, that enables the GPU to reduce DRAM bandwidth demands, making use of lossless compression techniques.

2.4.4.2 Shared memory atomics

Maxwell introduces native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap (CAS), which can be used to implement other atomic functions with reduced overhead compared to the Fermi and Kepler methods. This should make it much more efficient to implement things

like list and stack data structures shared by the threads of a block. [21].

Speeding up the code

3

3.1 Introduction

3.1.1 Image Registration

Image registration is a technology used to calculate a common frame of reference to a set of images that have been obtained at different times, from different points of view or from multiple devices [4]. A visual representation can be seen in figure 3.1.

To carry out this procedure, one of the images is taken as reference (target image) and the rest (floating images) are applied a series of geometric transformations so that certain points of an image fit with the corresponding points of another one.

The image registration is not an end itself, but used to enrich the information you already have, and it is applied to a large number of fields such as medicine, geophysics, robotics, etc.

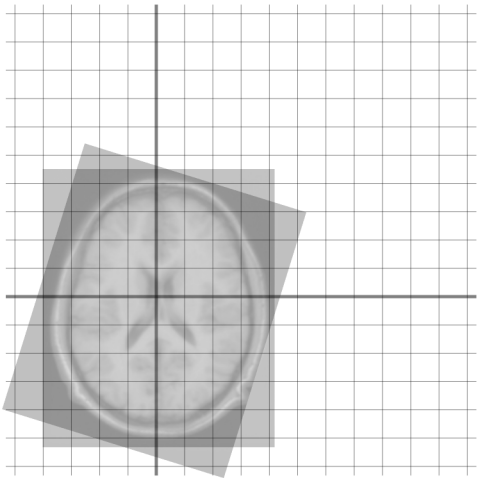

Figure 3.1: *Image registration.*

3.1.1.1 Operating mode

There are several methodologies (multimodal, template, viewpoint and temporary registration) that allow you to carry out image registration. But, regardless of the type of problem in which it is going to be applied, any image registration algorithm use the following components:

1. Similarity measure: It allows you to calculate the degree of similarity between two images.

2. Transformation function: It defines the way in which an image can be deformed. This function would make a transformation per each concrete sequence of parameters specified.

and it is computed as follows:

First Distinctive elements of images are extracted and matchings established between them.

Second Based on the matchings, the parameters of the transformation function are calculated.

Third The image to be registered is then transformed and the similarity to the reference image is calculated.

This process will be repeated iteratively while trying to maximize the value of similarity between images.

3.2 Starting point for optimizations

In spite of researchers are actively working on these techniques, its usage in clinical practice is still limited. Although the reasons may be different, in this project we are interested in performance. Applications became impractical when computing times could be extended from seconds to hours.

As a developer, the only option anyone has to carry out image registration without the need to write their own algorithm is to use a tool that provides that functionality. The one we are going to use during this project is called RNiftyReg and can be used through the R computing language.

NiftyReg, the application in which RNiftyReg based its source code, is a open-source software focused on medical image registration. It has been mainly developed by members of the Translational Imaging Group together with the Centre for Medical Image Computing at University College London, in United Kingdom. In addition to this, NiftyReg provides two versions of its algorithms, a CPU-based and a GPU-based implementation.

We have the source code of the applications mentioned before, RNiftyReg and NiftyReg. In addition to this, we also have a set of 23 magnetic resonance images. Can be seen its properties in the appendix A.1.

Through this chapter, I am going to describe the stages by which this project has passed until its final version.

3.3 Adding GPU support to RNiftyReg

R modules are add-ons that allow, in the same way a library does, to add functionality to the language. These modules are written using the R language and C/C++ code optionally. This is the way in which NiftyReg code is included inside RNiftyReg. However, the latter does not contain the whole application, so there are some features that are not available, such as GPU acceleration.

A module like RNiftyReg consists of a part of R code, written in the form of functions that would be called from a program written in R. On the other hand, it consists of C/C++ code that can be called from the functions mentioned before. Among several possible ways to add support for GPU module, we finally decided to modify the RNiftyReg C/C++ code so that it could call an external NiftyReg implementation already compiled and ready to run.

After finishing all these changes, an RNiftyReg with GPU acceleration was ready. In the figure 3.2 it is possible to see remarkable differences between execution times.

3.4 Optimizing NiftyReg

3.4.1 First iteration: Identifying the problem

Depending on what we want to optimize, we have to use certain tools to help us Identify the problem. In the case of this project, I have focused on enhancing the slowest piece of CPU code I could find inside NiftyReg.

Due to the fact that our aim is to find a piece of code that runs slowly in our host machine, the tool in which I have relied on is the CPU profiling framework called Valgrind. It provides you information about the execution of an application, including the rate of time each function takes to run.

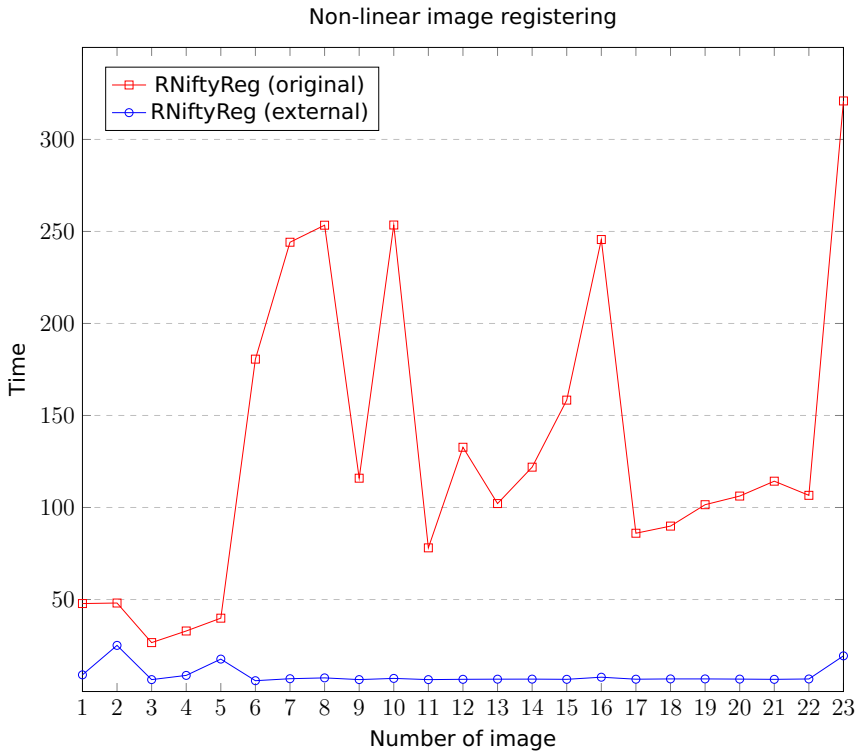

Figure 3.2: *Comparison between the original RNiftyReg and our version of RNiftyReg*

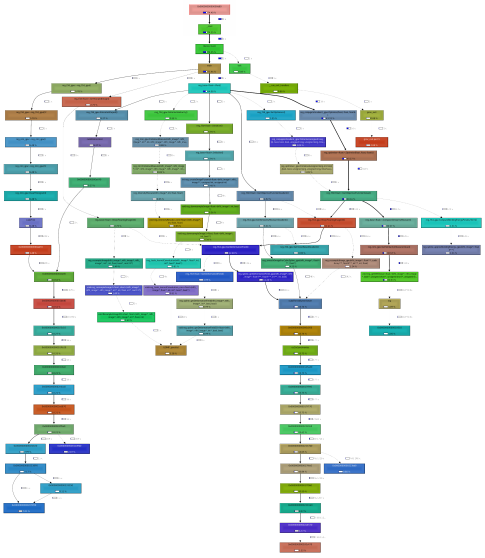

Figure 3.3: *The whole graph.*

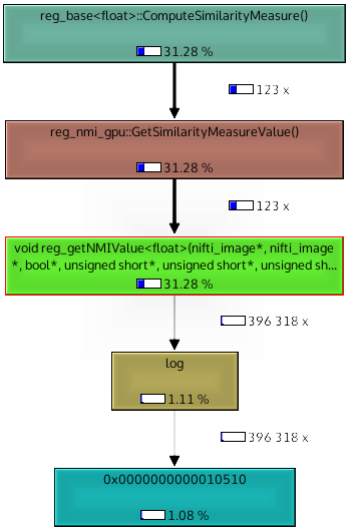

Figure 3.4: *Callgraph branch.*

That said, we can see the result of Valgrind in the figures 3.3 and 3.4. On the left side the whole call-graph of NiftyReg appears, while on the right side a concrete part of the call-graph has been zoomed in for a closeup.

Taking into account that the overall time each function takes also includes the time spent in auxiliary functions, the one with a maximum difference between this and its called functions is the slowest. Therefore, in our case, this is the function called reg_getNMIValue().

3.4.2 Second iteration: reg_getNMIValue() optimizations

reg_getNMIValue() is the method in charge of computing the mutual information, that is, the quantity of information that a pair of images are sharing.

3.4.2.1 Histogram smoothing

One of the critical parts of the calculation of this value consist of smoothing the values of a histogram. Its CPU implementation can be seen in Figure 3.5.

The histogram is smoothed in two different ways, and this is why there are two very similar pieces of code inside the reg_getNMIValue() method. Because of the similarities among these codes, from now onwards we will only explain in this report the work done in one of them. However, the other one will experience similar

```
...
for(int f=0; f<floatingBinNumber; ++f)
{
    for(int r=0; r<referenceBinNumber; ++r)
    {
        double value=0.0;
        int index = r-1;
        double *ptrHisto = &jointHistoProPtr[index+referenceBinNumber*f];

        for(int it=0; it<3; it++)
        {
            if(-1<index && index<referenceBinNumber)
            {
                value += *ptrHisto * kernel[it];
            }
            ++ptrHisto;
            ++index;
        }
        jointHistoLogPtr[r+referenceBinNumber*f] = value;
    }
}
...
```

Figure 3.5: *Original CPU code.*

changes.

The first step taken was the creation of a naive implementation of the algorithm to speed up. In this first version, we did not emphasize on creating a quick and efficient version, but to create an equivalent piece of code. After finishing this step, the execution time of NiftyReg was measured obtaining an expected speed-up value:

$$S_0 = \frac{T_{GPUOLD}}{T_{GPU0}} = \frac{15.53}{15.81} = 0.98$$

where T_{GPUOLD} is the time taken by the orignal version of NiftyReg, and T_{GPUn} the time taken by the $n-th$ optimization.

There exist a loss of performance, so our aim after this measurement was to improve our GPU algorithm. The piece of code is not computationally intensive, so the main problem must be related to the time wasted while accessing to memory.

It can be seen that every thread in that piece of program collect data from global memory three times (except from the threads in the edge of the histogram).

The collected data from one thread is also going to be collected by the previous and following threads. It is easy to realise that this data can be shared between them, avoiding the majority of accesses to the farthest memory.

Different types of memory are available to use through CUDA, each one with its own benefits and drawbacks. In case of trasfer speed, shared memory is around 500 times faster than global memory, so minimize the use of the latter is recommended.

To solve this problem, we have introduced shared memory in our kernel. For the sharedHistoPro variable, the memory was given a size equal to the number of elements to process plus two. These two extra memory positions were added because of the way the image is going to be smoothed.

To calculate an element n, a thread access to the elements $n-1$, n and $n+1$, so for these accesses to be efficient, they must be coalescing. In our case, the use of shared memory helps us avoiding part of the problem of non-coalesced access. The order in which the parameters 'f' and 'r' are asigned to threads, achieved that those who were consecutive, get access to consecutive global memory positions.

The measured speed-up accomplished is:

$$S_1 = \frac{T_{GPUOLD}}{T_{GPU1}} = \frac{15.53}{12.97} = 1.19$$

The latest improvement made to this algorithm was to add the #prama unroll directive, which allows to unroll a loop. The improvement accomplished here is not as obvious as the other improvements. However, after multiple executions, it can be seen that there exist a small speed-up:

$$S_2 = \frac{T_{GPUOLD}}{T_{GPU2}} = \frac{15.53}{12.90} = 1.20$$

After achieving this speed-ups, we can confirm we were right when we assumed the main problem of this algorithm was the way in which it access to data. The final version of the CUDA accelerated code can be seen in the figure 3.6.

3.4.2.2 Histogram smoothing (a failed alternative)

Addition to the process described above, another alternative implementation was attempted unsuccessfully.

It was tried that each thread had to pick a single value from global memory. Thus, there would be one thread for each iteration of the inner loop, which after

collecting its data, would add it to the corresponding position of shared memory. To perform this task it is necessary to use atomic operations.

After several attempts to optimize it, the computing times obtained were far superior compared to those obtained from the original GPU version. That is why this implementation was discarded in favour of another one for which was not necessary neither the use of atomic operations nor many global memory accesses.

3.4.2.3 Histogram normalization

As the final work done in this project, we have implemented two kernels responsible for normalizing the smoothed histogram.

The first of these kernels is responsible for making a reduction of the histogram, from which we get the largest element. The second kernel divides each element of the histogram by the value in the previous step. The implementation can be seen in the figure 3.7.

In spite this last optimization is smaller than the other explained before, it really worth it. After measuring the last of our algorithms, the speed-up achieved is:

$$S_3 = \frac{T_{GPUOLD}}{T_{GPU3}} = \frac{15.53}{11.2} = 1.38$$

3.5 About the achieved results

The progress achieved can be seen applied to the rest of the images we have in the figure 3.8. There, it may be noticed that in most cases, the difference among our version of RNfiftyReg and one with GPU acceleration is constant.

It seemed strage, so after a search we discovered that histograms have a constant size. Therefore, regardless of size of the input/output images, the computational load of reg_getNMIValue() is always the same.

It is important to highlight that results computed in GPU and CPU may differ, even when we had not changed NiftyReg. Therefore, it is logical that our changes also differ from the original GPU solution. This difference in results can be measured using Root-Mean-Square Difference (RMS).

When calculating the original GPU version RMS error with respect to CPU, we obtein a value of 45.28. When comparing GPUs versions, it has a value of 28.68.

Despite it is not a remarkable error, there are two reasons why these errors appear:

1. CPU operations are not strictly limited to 0.5 ulp (unit of least precision), so sequences of operations can be more accurate in CPU due to extended precision ALUs.
2. Floating-point arithmetic is not associative.

```
__global__ void reg_smoothJointHistogramAlongReferenceAxis(double *↩
    jointHistoLogPtr,double *jointHistoProPtr, double *kernel, unsigned ↩
    int floatingBinNumber, unsigned int referenceBinNumber)
{
    extern __shared__ double sharedHistoPro[];
    __shared__ double sharedKernel[3];

    int f = blockIdx.y;
    int r = threadIdx.x + blockIdx.x * blockDim.x;

    if ( !(f < floatingBinNumber && r < referenceBinNumber)) return;

    int index = r - 1;
    int arrayAddress = index + referenceBinNumber * f;
    sharedHistoPro[threadIdx.x] = jointHistoProPtr[arrayAddress];
    sharedHistoPro[threadIdx.x+2] = jointHistoProPtr[arrayAddress+2];
    __syncthreads();

    double value = 0.0;
    const int MAX_IT = 3;
    #pragma unroll
    for (int it = 0; it < MAX_IT; it++)
    {
        if (-1 < index && index < referenceBinNumber)
        {
                value += sharedHistoPro[threadIdx.x + it] * sharedKernel[↩
                    it];
        }
        ++arrayAddress;
        ++index;
    }
    jointHistoLogPtr[r + referenceBinNumber * f] = value;
}
```

Figure 3.6: *Smoothing algorithm implementation.*

```
template <unsigned int blockSize, bool nIsPow2>
__global__ void reg_reduction(double *g_idata, double *g_odata, short ↩
    unsigned int n)
{
    double *sdata = SharedMemory<double>();
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    double sum = 0;
    while (i < n)
    {
        sum += g_idata[i];

        if (nIsPow2 || i + blockSize < n)
            sum += g_idata[i+blockSize];

        i += gridSize;
    }

    // each thread puts its local sum into shared memory
    sdata[tid] = sum;
    __syncthreads();

    // do reduction in shared mememory
    if ((blockSize >= 1024) && (tid < 512))
    {
        sdata[tid] = sum = sum + sdata[tid + 512];
    }
    ...
    if ((blockSize >=   2) && ( tid <  1))
    {
        sdata[tid] = sum = sum + sdata[tid +  1];
    }
    __syncthreads();

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sum;
}

__global__ void reg_div(double *g_data, unsigned dataLen, double n)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < dataLen)
    {
        g_data[i] /= n;
    }
}
```

Figure 3.7: *Kernels involved in histogram normalization.*

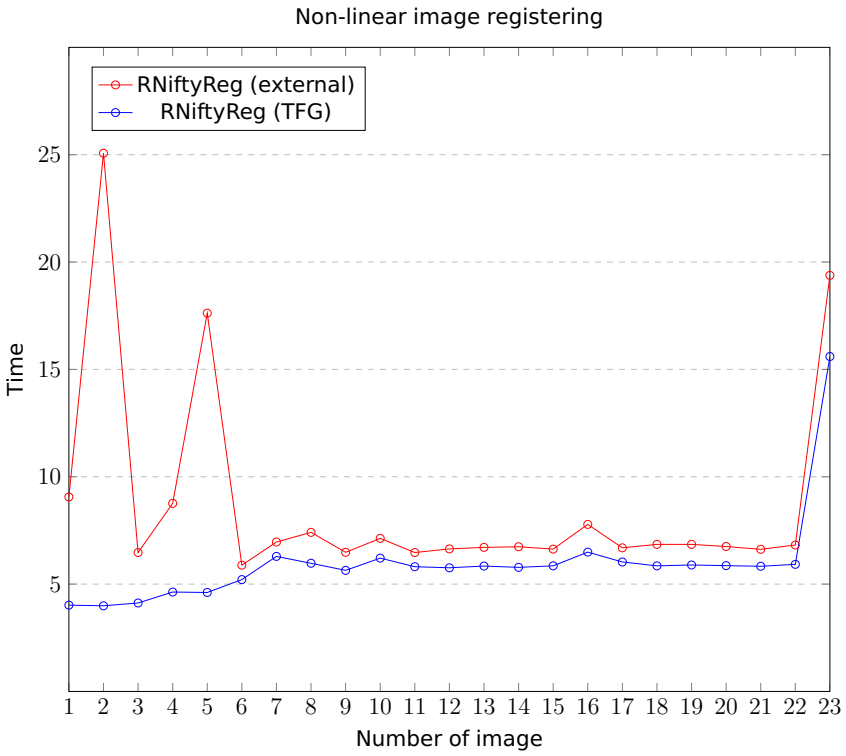

Figure 3.8: *Comparison between two versions of RNiftyReg: the original version with GPU acceleration, and the version modified in this project.*

Conclusions

4

This work has allowed to carry out the improvement of a medical image registration application. We have made improvements in two different algorithms, one of them focused on smoothing a histogram and the other one on normalizing a histogram.

The first of the algorithms that we have improved was a memory bound one, so the following hints were taken into account in the process of enhancing it:

1. Reduction of the number of memory accesses, prioritizing the use of faster memories when possible (registers and shared memory), while minimizing the use of more distant memory (global memory).
2. Properly access to memory, meaning that access should be grouped in sequences, which should also follow a concrete access pattern to take advantage of parallelism.

As a result of applying the points mentioned above on the image 23 of our data set, we have accomplished an speed-up value of 1,19x, in contrast to 0,98x from the naive implementation. In addition, a second optimization based on loop unrolling was performed, improving the speed-up to 1,20x.

Changes made on the second algorithm have allowed the application to achieve an speed-up of 1,38x.

In order to deploy parallelism, have been selected a suitable grid and blocks of threads size.

Finally, we can conclude that the quality of the improvements achieved over this project depends on the point of view. On one hand, if we compare our results to the GPU version NiftyReg, accelerations around 1,38x may not be particularly remarkable. Although there exist an improvement, the fact that this application was a GPU-accelerated one has not helped to highlight our enhancements. On the other hand, if we look at the speed-up obtained from the point of view of a RNiftyReg user,

a greater than 27x improvement can be considered quite good, since the only thing that allowed this module was run CPU versions of NiftyReg algorithms.

4.1 Conclusiones

Este trabajo ha permitido llevar a cabo la mejora de una aplicación destinada al registro de imágenes en el campo de la medicina. Se han llevado a cabo mejoras sobre dos algoritmos disintos, uno encargado de suavizar un histograma y el otro de normalizarlo.

El primero de los algoritmos que nos hemos encontrado era intensivo en memoria, por a la hora de mejorar el rendimiento ha sido muy útil:

1. Reducir el número de accesos a la memoria, priorizando el uso de memorias más rápidas cuando fuese posible (registros y memoria compartida), a la vez que se minimiza el uso de la memoria más lejana (memoria global).
2. Acceder de forma adecuada a la memoria, entendiendo que los accesos deben agruparse en secuencias, que además deben seguir un patrón predeterminado para aprovechar el paralelismo a la hora de acceder a los datos.

Como resultado de aplicar los puntos citados anteriormente sobre la imágen 23 de nuestro conjunto de datos, se ha conseguido pasar de una aceleración de 0,98x a 1,19x. Además, una segunda optimización basada en el desenrrollado de bucles ha subido la aceleración a 1,20x.

Los cambios aplicados sobre el segundo de los algoritmos, han permitido a la aplicación alcanzar una aceleración de 1,38x.

No menos importante ha sido desplegar en la medida de lo posible el paralelismo, teniendo que seleccionar para ello un tamaño adecuado de malla (grid) y bloques de hebras a la hora de optimizar el código en GPU.

Para terminar, se puede concluir que la calidad de las mejoras obtenidas a lo largo de este proyecto depende del punto de vista tomado. Por un lado, si miramos la aceleración con respecto a la versi’on GPU de NiftyReg, aceleraciones en torno a 1,38x pueden no ser especialmente significativas. A pesar de que la mejora existe, el hecho de que esta aplicación ya contase con aceleración por GPU no ha ayudado a que los resultados puedan destacar. Por otro lado, si miramos la aceleraciones obtenidas desde el punto de vista de un usuario de RNiftyReg, una mejora superior a 27x en algunos casos, se puede considerar bastante buena, ya que lo único que permitía este módulo era ejecutar las versiones CPU de los algoritmos de NiftyReg.

Appendix

	File name	x	y	z	total
1	20110225_153840t1sesags002a001	256	256	20	1310720
2	20120302_092511T1SEs201a1002	256	256	22	1441792
3	20120316_113132T1SEGADs901a1009	288	288	22	1824768
4	20120515_184622HIDROCEFALIAs002a1001	512	512	20	5242880
5	20120524_201906HIDROCEFALIAs002a1001	512	512	20	5242880
6	20120515_172040HIDROCEFALIAs007a1001	256	256	130	8519680
7	20130909_191805WIPsT1W3DTFESENSEs301a1003	256	256	175	11468800
8	20120515_172040HIDROCEFALIAs004a1001	512	512	53	13893632
9	20120316_113132NAVEGADORs701a1007	288	288	170	14100480
10	20120515_184622HIDROCEFALIAs004a1001	512	512	56	14680064
11	20121220_174920WIPsT1W3DTFESENSEs901a1009	288	288	185	15344640
12	20130429_210336WIPsT1W3DTFESENSEs301a1003	288	288	185	15344640
13	20130606_175455WIPsT1W3DTFESENSEs301a1003	288	288	185	15344640
14	20140221_142759WIPsT1W3DTFESENSEs601a1006	288	288	185	15344640
15	20140303_110039WIPsT1W3DTFESENSEs301a1003	288	288	185	15344640
16	20120524_201906HIDROCEFALIA4s005a1001	512	512	60	15728640
17	20090713_160720T13DISOTROPICOSENSEs301a...	288	288	190	15759360
18	20090907_172454T13DISOTROPICOSENSEs301a...	288	288	190	15759360
19	20090917_165550T13DISOTROPICOSENSEs301a...	288	288	190	15759360
20	20100413_173204T13DISOTROPICOSENSEs301a...	288	288	190	15759360
21	20100608_190105T13DISOTROPICOSENSEs301a...	288	288	190	15759360
22	20100706_193757T13DISOTROPICOSENSEs301a...	288	288	190	15759360
23	20120316_113132T1SEGADs801a1008	1024	1024	60	62914560

Tabla A.1: *Images metadata: file name, high, width, depth and total number of voxels.*

Bibliography

A

[1] Lisa Gottesfeld Brown. A survey of image registration techniques. ACM Computing Surveys 24:325–376, 1992.

[2] Chris McClanahan. History and Evolution of GPU Architecture. 2010.

[3] Christos Kyrkou. Stream Processors and GPUs: Architectures for High Performance Computing. Unknown.

[4] Mark Holden Derek L G Hill, Philipp G Batchelor and David J Hawkes. Medical image registration. *INSTITUTE OF PHYSICS PUBLISHING*, pages R2–R4, 2000.

[5] Mark Harris. Introduction to CUDA C, 2013.

[6] Ian Buck. *Stream Computing on Graphics Hardware*. PhD thesis, September 2006.

[7] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D Medical Image Registration CUDA Software with Genetic Programming. *Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO 2014)*, pages 951–958, 2014.

[8] Mark Harris. Maxwell: The Most Advanced CUDA GPU Ever Made, 2014. URL `http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`.

[9] Marc Modat, Zeike A. Taylor, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Sebastien Ourselin. Fast free-form deformation using the normalised mutual information gradient and graphics processing units. *Med Phys*, pages 278–284, 2010.

[10] Modat, M., Cash, D. M., Daga, P., Winston, G. P., Duncan, J. S., and Ourselin, S. Global image registration using a symmetric block-matching. *JOURNAL of Medical Imaging*, 1(2):024003–024003, 2014.

[11] Modat, M., Ridgway, G. R., Taylor, Z. A., Lehmann, M., Barnes, J., Hawkes, D. J., Fox, N. C., et al. Fast free-form deformation using graphics processing units. *Computer Methods And Programs In Biomedicine*, 98(3):278–284, 2010.

[12] NVIDIA Corporation. NVIDIA GeForce 8800 GPU architecture overview. Technical report, November 2006.

[13] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU architectural overview. Technical report, May 2008.

[14] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi Whitepaper. 2009.

[15] NVIDIA Corporation. NVIDIA GF100 Whitepaper. Technical report, 2010.

[16] NVIDIA Corporation. NVIDIA Tesla KSeries Data Sheet, October 2012.

[17] NVIDIA Corporation. TESLA K20X GPU Accelerator Board Specification. Technical report, July 2013.

[18] NVIDIA Corporation. TESLA K20 GPU Accelerator Board Specification. Technical report, July 2013.

[19] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210 Whitepaper. Technical report, 2014.

[20] NVIDIA Corporation. CUDA C Programming Guide, 2015. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[21] NVIDIA Corporation. NVIDIA GeForce GTX 980 Whitepaper. Technical report, 2015.

[22] Stephan Soller. GPGPU origins and GPU hardware architecture. 2011.

[23] UC London Translational Imaging Group. NiftyReg: Open-source software for efficient medical image registration, 2015. URL `http://cmictig.cs.ucl.ac.uk/research/software/22-niftyreg`.

[24] Barbara Zitová and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21:977–1000, 2003.